Real-time edge detection in camera viewport

This article explains how to optimise an image edge detection algorithm so that it can be used for "real-time" previewing of video in the camera's viewport.

27 May 2012

Note: This is an entry in the PureView Imaging Competition 2012Q2

Introduction

Image editing algorithms that take several seconds to complete are acceptable for Instagram-like static image editing, where quality is more important than speed (see Image editing techniques and algorithms using Qt for examples). However these same algorithms are not suitable for real-time usage (e.g. display in the camera's viewport) as they would result in visibly poor frame rates.

This article explains how to optimize an edge detection algorithm for calculation speed (possibly sacrificing some level of "quality") so that it can be applied to a video feed in real-time. The hope is that you will be able to apply same techniques for optimising other algorithms.

Camera viewport technical details

Before starting to work let's overview how we should handle Camera in Qt in order to manipulate viewport. For this task we will need to use QAbstractVideoSurface. You have to subclass this class and implement some specific methods. In `present()` method you get frame data that you can convert to image (QImage class) and draw where you want to using painter (QPainter class). Please note that there is a problem in Meego with frame data; this is covered in MeeGo Camera VideoSurface manipulation (we will return to MeeGo in this article as well).

Edge detection optimization

Below is an implementation of an edge detection algorithm (Sobel operator) for RGB image. The implementation has purposefully been written to be inefficient so we can demonstrate how it can be improved.

```
function calcLuma(QRgb val)
{
    int luma = (int)(0.2126 * (val >> 16 & 0xff) + 0.7152 * (val >> 8 & 0xff) + (0.0722 * (val & 0xff)));
    return luma
}

...

QImage newImage = QImage(image.width(), image.height(), QImage::Format_RGB32)

for (int y = 1; y < image.height()-1; y++)
    for (int x = 1; x < image.width()-1; x++)
    {
    // calculate luma (brightness) for points around this one
    int lxm1ym1 = calcLuma(image.pixel(x-1, y-1));
    int lxm1ym0 = calcLuma(image.pixel(x-1, y  ));
    int lxm1yp1 = calcLuma(image.pixel(x-1, y+1));
    int lxm0ym1 = calcLuma(image.pixel(x, y-1));
    int lxm0yp1 = calcLuma(image.pixel(x, y+1));
    int lxp1ym1 = calcLuma(image.pixel(x+1, y-1));
    int lxp1ym0 = calcLuma(image.pixel(x+1, y-1));
    int lxp1yp1 = calcLuma(image.pixel(x+1, y-1));

    // calculate gradients
    int dx = - lxm1ym1 - lxm1ym0 * 2 - lxm1yp1 + lxp1ym1 + lxp1ym0 * 2 + lxp1yp1;
    int dy = - lxm1ym1 - lxm0ym1 * 2 - lxp1ym1 + lxm1yp1 + lxp0ym1 * 2 + lxp1yp1;


    // calculate gradient magnitude
    int mag = qSqrt(dx*dx+dy*dy);
    if (mag > 255)
        mag = 255;

    // set pixel
    newImage.setPixel(x, y, mag + mag << 8 + mag << 16);
    }
```

This code will work but it is really slow and inefficient. So let's see what we could do better.

The first problem is that we shouldn't use `pixel()` and `setPixel()` functions. They are inefficient because multiplication is used to calculate position of pixel - internally it most probably look like this: `y * bytes_per_line + x * bytes_per_pixel`. It is much better to use `bits()` and `scanLine()` functions to get position of all image or position of line in memory respectively. In this case we will use `bits()` function to get position of first line and will get address of next line by adding `bytesPerLine()` to previous line (notice that we are iterating over `lines(y)` in our algorithm). This way we can replace `image.pixel(x,y)` functions with something like this:

```
uchar* lineAddress = image.bits();
....
int lxm1ym1 = calcLuma((int)*(lineAddress - bytes_per_line + (x-1) * 4));
```

This code will pass 4 bytes to `calcLuma()` function from position (x-1, y-1) because `-bytes_per_line = y-1.` x must be multiplied by 4 because we are working with RGB data and each pixel takes 4 bytes. Again since we are iterating over x as well instead of multiplying by 4 we can calculate next x value

(C) Copyright Nokia 2013. All rights reserved.

by adding 4 (again 4 bytes per pixel) to it. Basically if we want to eliminate using `pixel()` and `setPixel()` functions from our code we should use image data directly.

The next inefficient thing in our code is that we are recalculating same luma (brightness) values much more times than necessary. E.g. we are calculating luma for point (3, 3) when we are calculating gradient magnitude for all 8 points around this point). Solution to this is to calculate luma points only for 3 lines that are currently relevant to us for first line. For subsequent lines we only have to calculate next line. E.g.: let's say we are working in line 1 and already know luma points for line 0, 1 and 2:

```
0 . . . . . <- known luma points
1 . . . . . <- known luma points
2 . . . . . <- known luma points
3 . . . . . <- unknown luma points
```

Now when we need to calculate gradient magnitudes for points in line 2 we just reuse luma points from lines 1 and 2 and only have to calculate luma points in line 3.

Here we can make another optimization with memory as well (we should save memory usage in mobile devices if we can and if it doesn't hurt performance). Since we are calculating luma points for lines around current line it is actually safe to write on top of current image. This way we don't need to create new image in memory and will save about 1Mb of memory.

Next thing to optimize is luma calculation formula: Y = 0.2126 * R + 0.7152 * G + 0.0722 * B. First problem with this formula is that it uses real numbers and real numbers arithmetics is slower on computers than integer arithmetics. Therefore we change our formula in this way: Y = (0.2126 * 256 * R + 0.7152 * 256 * G + 0.0722 * 256 * B) / 256 = (54 * R + 183 * G + 18 * B) >> 8 - shift operation is faster than division operation on binary CPU what are majorities of CPUs nowadays and we now have formula that deals with integer numbers. And we can optimise this even more: R, G and B values are from range 0 to 255 therefore we can precalculate multiplication operations into arrays and our formula will be simple as (lumaR[R] + lumaG[G] + lumaB[B]) >> 8.

Note: In MeeGo Harmattan we have image data with luma already and it is much more efficient to use that data directly. In code sample that is attached to this article luma data is used directly in MeeGo Harmattan case.

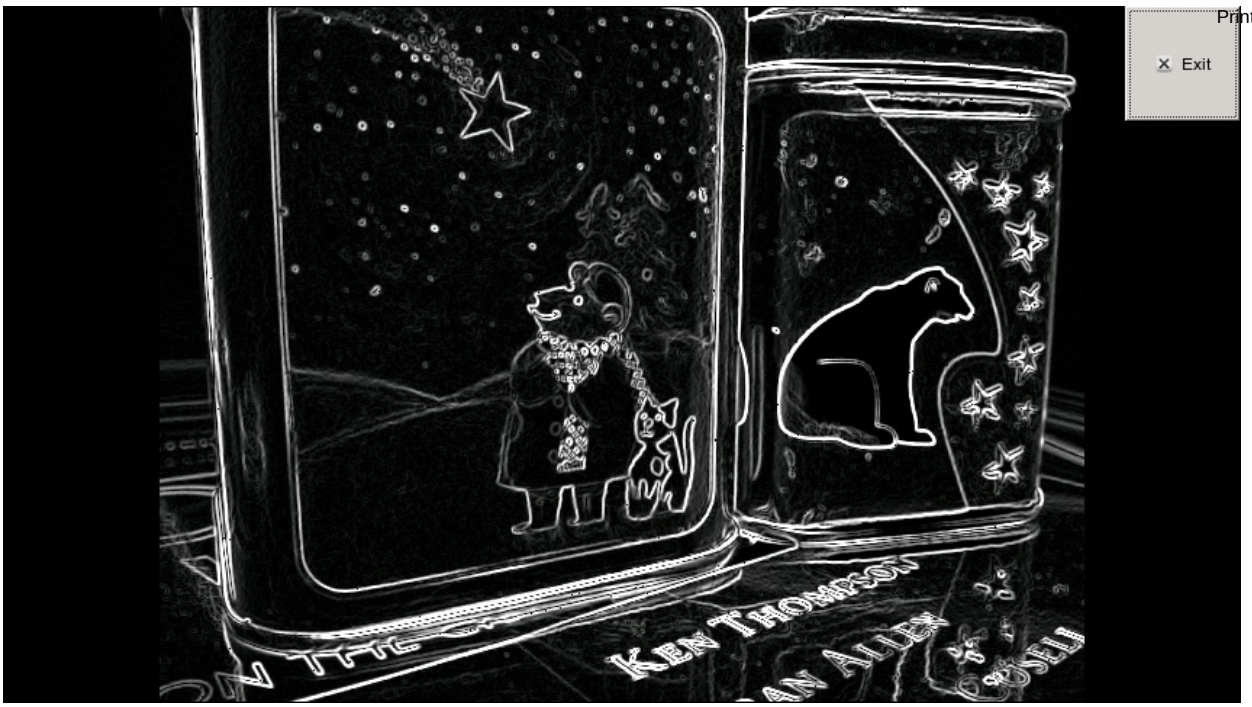
The last thing to solve is square root calculation - it is really expensive operation. Again we can optimize by precalculating values here: sqrt(x*x+y*y) will always be more than 255 when x or y is more than 255. In such case we need to create table of size 256x256 and precalculate square roots. Try using sqrt (`qSqrt()` function) instead of precalculated values and you will notice that view is lagging from about 0.5 to 2 seconds (depends on your phone's CPU). Here is MeeGo Harmattan implementation for precalculating square roots and using them:

```
    for (int y = 0; y < 255; y++)
        for (int x = 0; x < y; x++)
            {
            int val = (int)qSqrt(x*x+y*y);
            val = val > 255 ? 255 : val;
            sqrt_table[x][y] = val;
            sqrt_table[y][x] = val;
            }
    ....


    unsigned char *dst = m_bits + m_width * 4;
    const unsigned char *src = source.bits() + m_width * 2;


    for (int y = 1; y < m_height-1; ++y) {
        unsigned char *d_x = dst + 4;
        const unsigned char *s_x = src + 2 + 1;
        dst = dst + m_width * 4; // four bytes per pixel
        src = src + m_width * 2; // two bytes per pixel

        for (int x = 1; x < m_width-1; ++x)
        {
            int gx = qAbs(-(*(s_x-m_width*2-2)) - *(s_x-2)*2 - *(s_x+m_width*2-2) + *(s_x-m_width*2+2) + *(s_x+2)*2 + *(s_x+m_width*
            int gy = qAbs(-(*(s_x-m_width*2-2)) - *(s_x-m_width*2)*2 - *(s_x-m_width*2-2) + *(s_x+m_width*2-2) + *(s_x+m_width*2)*2

            int g;
            // Try using this instead of sqrt_table to see performance penalty
            // g = (int)qSqrt(gx*gx+gy*gy);
            // g = (g > 255 ? 255 : g);

            if (gx > 255 || gy > 255)
                g = 255;
            else
                g = sqrt_table[gx][gy];

            *d_x = g;
            *(d_x+1) = g;
            *(d_x+2) = g;
            *(d_x+3) = 0;

            d_x+=4;
            s_x+=2;
        }
    }
```

That's actually it: we have eliminated square root extraction and all multiplication operations from our algorithm and it now can be used in real-time. Download sample application from here to see complete algorithm: Media:Camera edge 2.zip

Here is what you might see using this app:

(C) Copyright Nokia 2013. All rights reserved.



And here are videos of this application in action. On Nokia N950:

The media player is loading...

On Nokia N8:

The media player is loading...

Nokia N950 screencast to show details that are not visible in filmed videos:

The media player is loading...

Summary

In this article I have demonstrated how to optimise algorithm (that uses complex calculations) for real-time image processing and displaying from camera.

(C) Copyright Nokia 2013. All rights reserved.

(C) Copyright Nokia 2013. All rights reserved.